

Semantics of Quantum Loops and Recursion

Louis LEMONNIER

University of Edinburgh



THE UNIVERSITY *of* EDINBURGH
informatics

Phiqus & co., LIX, École polytechnique. 31th October 2024

```
def fact(n):  
    if n == 0:  
        return 1  
    else:  
        return (n*fact(n-1))
```

```
let rec fact (n:int) =  
    if n=0 then 1  
    else n * fact (n-1)
```

Programming languages

```
fact :  
    movq    $0, %rax  
    je     .case_zero  
    movq    %rdi, %rax  
    pushq  %rax  
    subq   $1, %rdi  
    call   fact  
    popq   %rbx  
    multq  %rbx, %rax  
    jmp    .else  
.case_zero :  
    movq   $1, %rax  
.else :  
    ret
```

$\lambda m. \lambda g. (\lambda x. g(xx)) (\lambda x. g(xx)) (\lambda p. \lambda a. \lambda b. pab)$
 $(\lambda n. n(\lambda x. (\lambda a. \lambda b. b)) (\lambda a. \lambda b. a)) (\lambda f. \lambda x. fx)$
 $(m(\lambda n. \lambda f. \lambda x. n(\lambda g. \lambda h. h(gf)) (\lambda u. x) (\lambda x. x)))$

Semantics

Comes from Ancient Greek, *to provide a meaning*.

Semantics

Comes from Ancient Greek, *to provide a meaning*.

Operational semantics: **formal** program execution $t \rightarrow t'$.

Semantics

Comes from Ancient Greek, *to provide a meaning*.

Operational semantics: **formal** program execution $t \rightarrow t'$.

For example, $(\lambda x.x + 2)3 \rightarrow$

Semantics

Comes from Ancient Greek, *to provide a meaning*.

Operational semantics: **formal** program execution $t \rightarrow t'$.

For example, $(\lambda x.x + 2)3 \rightarrow 3 + 2 \rightarrow$

Semantics

Comes from Ancient Greek, *to provide a meaning*.

Operational semantics: **formal** program execution $t \rightarrow t'$.

For example, $(\lambda x.x + 2)3 \rightarrow 3 + 2 \rightarrow 5$.

Semantics

Comes from Ancient Greek, *to provide a meaning*.

Operational semantics: **formal** program execution $t \rightarrow t'$.

For example, $(\lambda x.x + 2)3 \rightarrow 3 + 2 \rightarrow 5$.

Denotational semantics: **function** $\llbracket t \rrbracket$, usual in mathematical terms.

Semantics

Comes from Ancient Greek, *to provide a meaning*.

Operational semantics: **formal** program execution $t \rightarrow t'$.

For example, $(\lambda x.x + 2)3 \rightarrow 3 + 2 \rightarrow 5$.

Denotational semantics: **function** $\llbracket t \rrbracket$, usual in mathematical terms.

Preserves composition: $\llbracket t; t' \rrbracket = \llbracket t' \rrbracket \circ \llbracket t \rrbracket$

Semantics

Comes from Ancient Greek, *to provide a meaning*.

Operational semantics: **formal** program execution $t \rightarrow t'$.

For example, $(\lambda x.x + 2)3 \rightarrow 3 + 2 \rightarrow 5$.

Denotational semantics: **function** $\llbracket t \rrbracket$, usual in mathematical terms.

Preserves composition: $\llbracket t; t' \rrbracket = \llbracket t' \rrbracket \circ \llbracket t \rrbracket$

Strong enough model if **relation** between operational and denotational:

$$t \simeq t' \text{ iff } \llbracket t \rrbracket = \llbracket t' \rrbracket .$$

Semantics

Comes from Ancient Greek, *to provide a meaning*.

Operational semantics: **formal** program execution $t \rightarrow t'$.

For example, $(\lambda x.x + 2)3 \rightarrow 3 + 2 \rightarrow 5$.

Denotational semantics: **function** $\llbracket t \rrbracket$, usual in mathematical terms.

Preserves composition: $\llbracket t; t' \rrbracket = \llbracket t' \rrbracket \circ \llbracket t \rrbracket$

Strong enough model if **relation** between operational and denotational:

$$t \simeq t' \text{ iff } \llbracket t \rrbracket = \llbracket t' \rrbracket .$$

- **prove** properties without running the code,
- compile the language and perform optimisations **safely**,
- (sometimes) help on which **features** can be added to the language.

A gentle walk through classical programming

There needs to be a corresponding semantics to **all programs**.

```
def f(n):  
    return (n+1)
```

A gentle walk through classical programming

There needs to be a corresponding semantics to **all programs**.

```
def f(n):  
    return (n+1)
```

$$\llbracket f \rrbracket = \begin{cases} \mathbb{N} & \rightarrow \mathbb{N} \\ n & \mapsto n + 1 \end{cases}$$

A gentle walk through classical programming

There needs to be a corresponding semantics to **all programs**.

```
def f(n):  
    return (n+1)
```

```
def f(n):  
    return (g(n)+1)
```

$$\llbracket f \rrbracket = \begin{cases} \mathbb{N} & \rightarrow \mathbb{N} \\ n & \mapsto n + 1 \end{cases}$$

A gentle walk through classical programming

There needs to be a corresponding semantics to **all programs**.

```
def f(n):  
    return (n+1)
```

```
def f(n):  
    return (g(n)+1)
```

$$\llbracket f \rrbracket = \begin{cases} \mathbb{N} & \rightarrow \mathbb{N} \\ n & \mapsto n + 1 \end{cases}$$

$$\llbracket f \rrbracket = \begin{cases} \mathbb{N} & \rightarrow \mathbb{N} \\ n & \mapsto \llbracket g \rrbracket (n) + 1 \end{cases}$$

A gentle walk through classical programming

There needs to be a corresponding semantics to **all programs**.

```
def f(n):  
    return (n+1)
```

$$\llbracket f \rrbracket = \begin{cases} \mathbb{N} & \rightarrow \mathbb{N} \\ n & \mapsto n + 1 \end{cases}$$

```
def f(n):  
    return (g(n)+1)
```

$$\llbracket f \rrbracket = \begin{cases} \mathbb{N} & \rightarrow \mathbb{N} \\ n & \mapsto \llbracket g \rrbracket (n) + 1 \end{cases}$$

```
def f(n):  
    while True:  
        pass  
    return (n+1)
```

A gentle walk through classical programming

There needs to be a corresponding semantics to **all programs**.

```
def f(n):  
    return (n+1)
```

$$\llbracket f \rrbracket = \begin{cases} \mathbb{N} & \rightarrow \mathbb{N} \\ n & \mapsto n + 1 \end{cases}$$

```
def f(n):  
    return (g(n)+1)
```

$$\llbracket f \rrbracket = \begin{cases} \mathbb{N} & \rightarrow \mathbb{N} \\ n & \mapsto \llbracket g \rrbracket (n) + 1 \end{cases}$$

```
def f(n):  
    while True:  
        pass  
    return (n+1)
```

$$\llbracket f \rrbracket = \begin{cases} \mathbb{N} & \rightarrow \mathbb{N} \\ n & \mapsto ??? \end{cases}$$

A gentle walk through classical programming

There needs to be a corresponding semantics to **all programs**.

```
def f(n):  
    return (n+1)
```

$$\llbracket f \rrbracket = \begin{cases} \mathbb{N} & \rightarrow \mathbb{N} \\ n & \mapsto n + 1 \end{cases}$$

```
def f(n):  
    return (g(n)+1)
```

$$\llbracket f \rrbracket = \begin{cases} \mathbb{N} & \rightarrow \mathbb{N} \\ n & \mapsto \llbracket g \rrbracket (n) + 1 \end{cases}$$

```
def f(n):  
    while True:  
        pass  
    return (n+1)
```

$$\llbracket f \rrbracket = \begin{cases} \mathbb{N} & \rightarrow \mathbb{N} \\ n & \mapsto ??? \end{cases}$$

We need to account for **non termination**!

A gentle walk through classical programming

There needs to be a corresponding semantics to **all programs**.

```
def f(n):  
    return (n+1)
```

$$\llbracket f \rrbracket = \begin{cases} \mathbb{N} & \rightarrow \mathbb{N} \\ n & \mapsto n + 1 \end{cases}$$

```
def f(n):  
    return (g(n)+1)
```

$$\llbracket f \rrbracket = \begin{cases} \mathbb{N} & \rightarrow \mathbb{N} \\ n & \mapsto \llbracket g \rrbracket (n) + 1 \end{cases}$$

```
def f(n):  
    while True:  
        pass  
    return (n+1)
```

$$\llbracket f \rrbracket = \begin{cases} \mathbb{N} & \rightarrow \mathbb{N} \\ n & \mapsto ??? \end{cases}$$

We need to account for **non termination**!

\perp means **undefined**.

A gentle walk through classical programming

There needs to be a corresponding semantics to **all programs**.

```
def f(n):  
    return (n+1)
```

```
def f(n):  
    return (g(n)+1)
```

```
def f(n):  
    while True:  
        pass  
    return (n+1)
```

$$\llbracket f \rrbracket = \begin{cases} \mathbb{N} & \rightarrow \mathbb{N} \\ n & \mapsto n+1 \end{cases}$$

$$\llbracket f \rrbracket = \begin{cases} \mathbb{N} & \rightarrow \mathbb{N} \\ n & \mapsto \llbracket g \rrbracket (n) + 1 \end{cases}$$

$$\llbracket f \rrbracket = \begin{cases} \mathbb{N} & \rightarrow \mathbb{N} \\ n & \mapsto ??? \end{cases}$$

We need to account for **non termination**!

\perp means **undefined**.

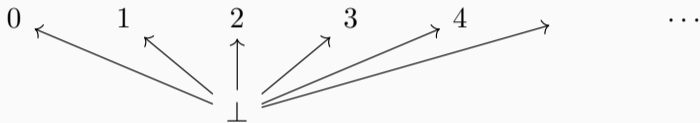
$$\llbracket f \rrbracket = \begin{cases} \mathbb{N} \cup \{\perp\} & \rightarrow \mathbb{N} \cup \{\perp\} \\ n & \mapsto \perp \end{cases}$$

Domain theory (example)

$\mathbb{N}_\perp \stackrel{\text{def}}{=} \mathbb{N} \cup \{\perp\}$ is a partial-ordered set with **nice** properties.

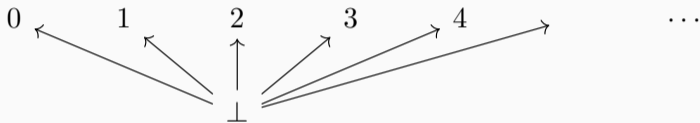
Domain theory (example)

$\mathbb{N}_\perp \stackrel{\text{def}}{=} \mathbb{N} \cup \{\perp\}$ is a partial-ordered set with **nice** properties.



Domain theory (example)

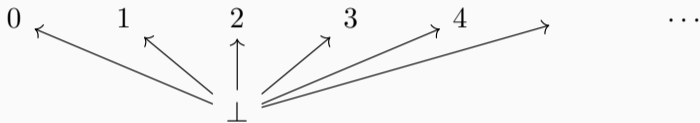
$\mathbb{N}_\perp \stackrel{\text{def}}{=} \mathbb{N} \cup \{\perp\}$ is a partial-ordered set with **nice** properties.



The order generalises to functions: $f \leq g$ if for all x , we have $f(x) \leq g(x)$.

Domain theory (example)

$\mathbb{N}_\perp \stackrel{\text{def}}{=} \mathbb{N} \cup \{\perp\}$ is a partial-ordered set with **nice** properties.

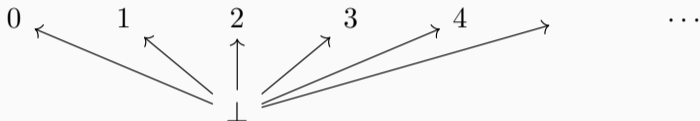


The order generalises to functions: $f \leq g$ if for all x , we have $f(x) \leq g(x)$.

The set $[\mathbb{N}_\perp \rightarrow \mathbb{N}_\perp]$ of **monotone** functions also has **nice** properties.

Domain theory (example)

$\mathbb{N}_\perp \stackrel{\text{def}}{=} \mathbb{N} \cup \{\perp\}$ is a partial-ordered set with **nice** properties.



The order generalises to functions: $f \leq g$ if for all x , we have $f(x) \leq g(x)$.

The set $[\mathbb{N}_\perp \rightarrow \mathbb{N}_\perp]$ of **monotone** functions also has **nice** properties.

How do we use it?

Interpretation of recursion

Define $F : [[\mathbb{N}_\perp \rightarrow \mathbb{N}_\perp] \rightarrow [\mathbb{N}_\perp \rightarrow \mathbb{N}_\perp]]$ as:

$$F(g)(n) = \begin{cases} 1 & \text{if } n = 0, \\ n \times g(n-1) & \text{else.} \end{cases}$$

Write $\perp : \mathbb{N}_\perp \rightarrow \mathbb{N}_\perp$ the constant function with output \perp .

$\perp : \mathbb{N}_\perp \rightarrow \mathbb{N}_\perp$ is:

$\perp \quad \perp \quad \perp \quad \perp \quad \perp \quad \perp \quad \perp \quad \dots$

Interpretation of recursion

Define $F : [[\mathbb{N}_\perp \rightarrow \mathbb{N}_\perp] \rightarrow [\mathbb{N}_\perp \rightarrow \mathbb{N}_\perp]]$ as:

$$F(g)(n) = \begin{cases} 1 & \text{if } n = 0, \\ n \times g(n-1) & \text{else.} \end{cases}$$

Write $\perp : \mathbb{N}_\perp \rightarrow \mathbb{N}_\perp$ the constant function with output \perp .

$\perp : \mathbb{N}_\perp \rightarrow \mathbb{N}_\perp$ is:

$\perp \quad \perp \quad \perp \quad \perp \quad \perp \quad \perp \quad \perp \quad \dots$

$F(\perp) : \mathbb{N}_\perp \rightarrow \mathbb{N}_\perp$ is:

$1 \quad \perp \quad \perp \quad \perp \quad \perp \quad \perp \quad \perp \quad \dots$

Interpretation of recursion

Define $F : [[\mathbb{N}_\perp \rightarrow \mathbb{N}_\perp] \rightarrow [\mathbb{N}_\perp \rightarrow \mathbb{N}_\perp]]$ as:

$$F(g)(n) = \begin{cases} 1 & \text{if } n = 0, \\ n \times g(n-1) & \text{else.} \end{cases}$$

Write $\perp : \mathbb{N}_\perp \rightarrow \mathbb{N}_\perp$ the constant function with output \perp .

$F(\perp) : \mathbb{N}_\perp \rightarrow \mathbb{N}_\perp$ is:

1 \perp \perp \perp \perp \perp \perp ...

$F(F(\perp)) : \mathbb{N}_\perp \rightarrow \mathbb{N}_\perp$ is:

1 1 \perp \perp \perp \perp \perp ...

Interpretation of recursion

Define $F : [[\mathbb{N}_\perp \rightarrow \mathbb{N}_\perp] \rightarrow [\mathbb{N}_\perp \rightarrow \mathbb{N}_\perp]]$ as:

$$F(g)(n) = \begin{cases} 1 & \text{if } n = 0, \\ n \times g(n-1) & \text{else.} \end{cases}$$

Write $\perp : \mathbb{N}_\perp \rightarrow \mathbb{N}_\perp$ the constant function with output \perp .

$F(F(\perp)) : \mathbb{N}_\perp \rightarrow \mathbb{N}_\perp$ is:

1 1 \perp \perp \perp \perp \perp ...

$F(F(F(\perp))) : \mathbb{N}_\perp \rightarrow \mathbb{N}_\perp$ is:

1 1 2 \perp \perp \perp \perp ...

Interpretation of recursion

Define $F : [[\mathbb{N}_\perp \rightarrow \mathbb{N}_\perp] \rightarrow [\mathbb{N}_\perp \rightarrow \mathbb{N}_\perp]]$ as:

$$F(g)(n) = \begin{cases} 1 & \text{if } n = 0, \\ n \times g(n-1) & \text{else.} \end{cases}$$

Write $\perp : \mathbb{N}_\perp \rightarrow \mathbb{N}_\perp$ the constant function with output \perp .

$F(F(F(\perp))) : \mathbb{N}_\perp \rightarrow \mathbb{N}_\perp$ is:

1 1 2 \perp \perp \perp \perp ...

$F(F(F(F(\perp)))) : \mathbb{N}_\perp \rightarrow \mathbb{N}_\perp$ is:

1 1 2 6 \perp \perp \perp ...

Interpretation of recursion

Define $F : [[\mathbb{N}_\perp \rightarrow \mathbb{N}_\perp] \rightarrow [\mathbb{N}_\perp \rightarrow \mathbb{N}_\perp]]$ as:

$$F(g)(n) = \begin{cases} 1 & \text{if } n = 0, \\ n \times g(n-1) & \text{else.} \end{cases}$$

Write $\perp : \mathbb{N}_\perp \rightarrow \mathbb{N}_\perp$ the constant function with output \perp .

$F(F(F(F(\perp)))) : \mathbb{N}_\perp \rightarrow \mathbb{N}_\perp$ is:

1 1 2 6 \perp \perp \perp ...

$F(F(F(F(F(\perp)))))) : \mathbb{N}_\perp \rightarrow \mathbb{N}_\perp$ is:

1 1 2 6 24 \perp \perp ...

Interpretation of recursion

Define $F : [[\mathbb{N}_\perp \rightarrow \mathbb{N}_\perp] \rightarrow [\mathbb{N}_\perp \rightarrow \mathbb{N}_\perp]]$ as:

$$F(g)(n) = \begin{cases} 1 & \text{if } n = 0, \\ n \times g(n-1) & \text{else.} \end{cases}$$

Write $\perp : \mathbb{N}_\perp \rightarrow \mathbb{N}_\perp$ the constant function with output \perp .

$F(F(F(F(F(\perp)))))) : \mathbb{N}_\perp \rightarrow \mathbb{N}_\perp$ is:

1 1 2 6 24 \perp \perp ...

$F(F(F(F(F(F(\perp))))))) : \mathbb{N}_\perp \rightarrow \mathbb{N}_\perp$ is:

1 1 2 6 24 120 \perp ...

Interpretation of recursion

Define $F : [[\mathbb{N}_\perp \rightarrow \mathbb{N}_\perp] \rightarrow [\mathbb{N}_\perp \rightarrow \mathbb{N}_\perp]]$ as:

$$F(g)(n) = \begin{cases} 1 & \text{if } n = 0, \\ n \times g(n-1) & \text{else.} \end{cases}$$

Write $\perp : \mathbb{N}_\perp \rightarrow \mathbb{N}_\perp$ the constant function with output \perp .

$F(F(F(F(F(\perp)))))) : \mathbb{N}_\perp \rightarrow \mathbb{N}_\perp$ is:

1 1 2 6 24 \perp \perp ...

$F(F(F(F(F(F(\perp))))))) : \mathbb{N}_\perp \rightarrow \mathbb{N}_\perp$ is:

1 1 2 6 24 120 \perp ...

The interpretation of the factorial is then the **limit** of $(F^n(\perp))_n$.

Interpretation of recursion

Define $F : [[\mathbb{N}_\perp \rightarrow \mathbb{N}_\perp] \rightarrow [\mathbb{N}_\perp \rightarrow \mathbb{N}_\perp]]$ as:

$$F(g)(n) = \begin{cases} 1 & \text{if } n = 0, \\ n \times g(n-1) & \text{else.} \end{cases}$$

Write $\perp : \mathbb{N}_\perp \rightarrow \mathbb{N}_\perp$ the constant function with output \perp .

$F(F(F(F(F(\perp)))))) : \mathbb{N}_\perp \rightarrow \mathbb{N}_\perp$ is:

1 1 2 6 24 \perp \perp ...

$F(F(F(F(F(F(\perp))))))) : \mathbb{N}_\perp \rightarrow \mathbb{N}_\perp$ is:

1 1 2 6 24 120 \perp ...

The interpretation of the factorial is then the **limit** of $(F^n(\perp))_n$.

A notion of **limit** linked to the order is necessary.

What about Quantum Programming?

Quantum channels: e.g. CPTN maps $\mathcal{L}(H_A) \rightarrow \mathcal{L}(H_B)$.

What about Quantum Programming?

Quantum channels: e.g. CPTN maps $\mathcal{L}(H_A) \rightarrow \mathcal{L}(H_B)$.

Löwner order: $f \leq g$ iff $g - f$ is positive.

What about Quantum Programming?

Quantum channels: e.g. CPTN maps $\mathcal{L}(H_A) \rightarrow \mathcal{L}(H_B)$.

Löwner order: $f \leq g$ iff $g - f$ is positive.

Admits a **nice** structure in the (so called) Heisenberg picture.

What about Quantum Programming?

Quantum channels: e.g. CPTN maps $\mathcal{L}(H_A) \rightarrow \mathcal{L}(H_B)$.

Löwner order: $f \leq g$ iff $g - f$ is positive.

Admits a **nice** structure in the (so called) Heisenberg picture.

The **undefined** function is interpreted as the constant map 0 (no need to add a \perp).

What about Quantum Programming?

Quantum channels: e.g. CPTN maps $\mathcal{L}(H_A) \rightarrow \mathcal{L}(H_B)$.

Löwner order: $f \leq g$ iff $g - f$ is positive.

Admits a **nice** structure in the (so called) Heisenberg picture.

The **undefined** function is interpreted as the constant map 0 (no need to add a \perp).

What about Quantum Programming?

Quantum channels: e.g. CPTN maps $\mathcal{L}(H_A) \rightarrow \mathcal{L}(H_B)$.

Löwner order: $f \leq g$ iff $g - f$ is positive.

Admits a **nice** structure in the (so called) Heisenberg picture.

The **undefined** function is interpreted as the constant map 0 (no need to add a \perp).

★ (almost) everything done similar to classical computing.

What about Quantum Programming?

Quantum channels: e.g. CPTN maps $\mathcal{L}(H_A) \rightarrow \mathcal{L}(H_B)$.

Löwner order: $f \leq g$ iff $g - f$ is positive.

Admits a **nice** structure in the (so called) Heisenberg picture.

The **undefined** function is interpreted as the constant map 0 (no need to add a \perp).

- ★ (almost) everything done similar to classical computing.
- ★ with probabilities (because of measurement).

What about Quantum Programming?

Quantum channels: e.g. CPTN maps $\mathcal{L}(H_A) \rightarrow \mathcal{L}(H_B)$.

Löwner order: $f \leq g$ iff $g - f$ is positive.

Admits a **nice** structure in the (so called) Heisenberg picture.

The **undefined** function is interpreted as the constant map 0 (no need to add a \perp).

- ★ (almost) everything done similar to classical computing.
- ★ with probabilities (because of measurement).
- ★ and a non duplication of terms because of no-cloning.

Now, let's compute reversibly!

[SabryValironVizzotto2018] Language for expressing (quantum) reversible programs.

The Reversible Case

[SabryValironVizzotto2018] Language for expressing (quantum) reversible programs.

The language is very expressive and has a **programming flavour**.

The Reversible Case

[SabryValironVizzotto2018] Language for expressing (quantum) reversible programs.

The language is very expressive and has a **programming flavour**.

Its classical counterpart is interpreted with **partial injective** functions.

The Reversible Case

[SabryValironVizzotto2018] Language for expressing (quantum) reversible programs.

The language is very expressive and has a **programming flavour**.

Its classical counterpart is interpreted with **partial injective** functions.

- ★ a **full** partial injection is a bijection.

The Reversible Case

[SabryValironVizzotto2018] Language for expressing (quantum) reversible programs.

The language is very expressive and has a **programming flavour**.

Its classical counterpart is interpreted with **partial injective** functions.

- ★ a **full** partial injection is a bijection.

Its quantum counterpart is interpreted as **contractive** linear maps.

The Reversible Case

[SabryValironVizzotto2018] Language for expressing (quantum) reversible programs.

The language is very expressive and has a **programming flavour**.

Its classical counterpart is interpreted with **partial injective** functions.

- ★ a **full** partial injection is a bijection.

Its quantum counterpart is interpreted as **contractive** linear maps.

- ★ a **full** contractive map is a unitary.

The Reversible Case

[SabryValironVizzotto2018] Language for expressing (quantum) reversible programs.

The language is very expressive and has a **programming flavour**.

Its classical counterpart is interpreted with **partial injective** functions.

- ★ a **full** partial injection is a bijection.

Its quantum counterpart is interpreted as **contractive** linear maps.

- ★ a **full** contractive map is a unitary.

Is it possible to interpret loop or recursion in this setting?

Quick stop: Recursion in Classical Reversibility

Suppose we have a syntax with:

- Only reversible functions,
- Inductive types,
- A fixpoint operator,
- A CbV operational semantics.

Quick stop: Recursion in Classical Reversibility

Suppose we have a syntax with:

- Only reversible functions,
- Inductive types,
- A fixpoint operator,
- A CbV operational semantics.

It is Turing-complete! [ChardonnetLemonnierValiron2024]

Quick stop: Recursion in Classical Reversibility

Suppose we have a syntax with:

- Only reversible functions,
- Inductive types,
- A fixpoint operator,
- A CbV operational semantics.

It is Turing-complete! [ChardonnetLemonnierValiron2024]

Given two sets A and B , the partial injections $[A \multimap B]$ have **nice** properties for fixpoints.

Quick stop: Recursion in Classical Reversibility

Suppose we have a syntax with:

- Only reversible functions,
- Inductive types,
- A fixpoint operator,
- A CbV operational semantics.

It is Turing-complete! [ChardonnetLemonnierValiron2024]

Given two sets A and B , the partial injections $[A \multimap B]$ have **nice** properties for fixpoints.

(The category of partial injective functions between sets is enriched in pointed dcpos.)

Quick stop: Recursion in Classical Reversibility

Suppose we have a syntax with:

- Only reversible functions,
- Inductive types,
- A fixpoint operator,
- A CbV operational semantics.

It is Turing-complete! [ChardonnetLemonnierValiron2024]

Given two sets A and B , the partial injections $[A \multimap B]$ have **nice** properties for fixpoints.

(The category of partial injective functions between sets is enriched in pointed dcpos.)

What about its quantum counterpart?

The Quantum Case

Reversible quantum operations are **subunitaries** (partial isometries) such as:

$$\begin{aligned} |0\rangle\langle 0| + |1\rangle\langle 1| &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} & |0\rangle\langle 0| &= \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} & |1\rangle\langle 1| &= \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \\ |+\rangle\langle 0| + |-\rangle\langle 1| &= \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix} & |+\rangle\langle 0| &= \begin{bmatrix} \frac{1}{\sqrt{2}} & 0 \\ \frac{1}{\sqrt{2}} & 0 \end{bmatrix} & |-\rangle\langle 1| &= \begin{bmatrix} 0 & \frac{1}{\sqrt{2}} \\ 0 & -\frac{1}{\sqrt{2}} \end{bmatrix} \\ |1\rangle\langle 0| + |0\rangle\langle 1| &= \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} & |1\rangle\langle 0| &= \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} & |0\rangle\langle 1| &= \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \end{aligned}$$

The Quantum Case

Reversible quantum operations are **subunitaries** (partial isometries) such as:

$$\begin{aligned} |0\rangle\langle 0| + |1\rangle\langle 1| &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} & |0\rangle\langle 0| &= \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} & |1\rangle\langle 1| &= \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \\ |+\rangle\langle 0| + |-\rangle\langle 1| &= \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix} & |+\rangle\langle 0| &= \begin{bmatrix} \frac{1}{\sqrt{2}} & 0 \\ \frac{1}{\sqrt{2}} & 0 \end{bmatrix} & |-\rangle\langle 1| &= \begin{bmatrix} 0 & \frac{1}{\sqrt{2}} \\ 0 & -\frac{1}{\sqrt{2}} \end{bmatrix} \\ |1\rangle\langle 0| + |0\rangle\langle 1| &= \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} & |1\rangle\langle 0| &= \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} & |0\rangle\langle 1| &= \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \end{aligned}$$

Arguably, **undefined** (\perp) in a (2-dim.) Hilbert space is the **zero** vector $\begin{bmatrix} 0 \\ 0 \end{bmatrix}$.

The Quantum Case

Reversible quantum operations are **subunitaries** (partial isometries) such as:

$$\begin{aligned} |0\rangle\langle 0| + |1\rangle\langle 1| &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} & |0\rangle\langle 0| &= \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} & |1\rangle\langle 1| &= \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \\ |+\rangle\langle 0| + |-\rangle\langle 1| &= \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix} & |+\rangle\langle 0| &= \begin{bmatrix} \frac{1}{\sqrt{2}} & 0 \\ \frac{1}{\sqrt{2}} & 0 \end{bmatrix} & |-\rangle\langle 1| &= \begin{bmatrix} 0 & \frac{1}{\sqrt{2}} \\ 0 & -\frac{1}{\sqrt{2}} \end{bmatrix} \\ |1\rangle\langle 0| + |0\rangle\langle 1| &= \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} & |1\rangle\langle 0| &= \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} & |0\rangle\langle 1| &= \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \end{aligned}$$

Arguably, **undefined** (\perp) in a (2-dim.) Hilbert space is the **zero** vector $\begin{bmatrix} 0 \\ 0 \end{bmatrix}$.

$$\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \text{ is less defined than } \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} = |0\rangle\langle 0| \text{ which is less defined than } \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

A disappointment

However, this order does not interact well with composition:

$$\begin{aligned} & \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \leq \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \\ \text{thus} \quad & \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix} \circ \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \circ \begin{bmatrix} \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{bmatrix} \leq \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix} \circ \begin{bmatrix} \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{bmatrix}, \\ \text{i.e.} \quad & 1/2 \leq 0. \end{aligned}$$

A disappointment

However, this order does not interact well with composition:

$$\begin{aligned} & \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \leq \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \\ \text{thus} \quad & \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix} \circ \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \circ \begin{bmatrix} \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{bmatrix} \leq \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix} \circ \begin{bmatrix} \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{bmatrix}, \\ \text{i.e.} \quad & 1/2 \leq 0. \end{aligned}$$

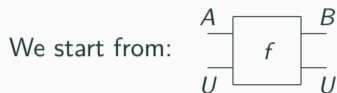
Theorem (No go theorem)

There is no such order on subunitary maps between Hilbert spaces.

Recursion through a Trace



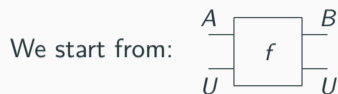
Recursion through a Trace



The recurring example: g :

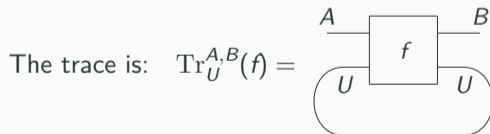
$$\left\{ \begin{array}{ll} \mathbb{N} \uplus (\mathbb{N} \times \mathbb{N}) & \rightarrow \mathbb{N} \uplus (\mathbb{N} \times \mathbb{N}) \\ \text{left } n & \mapsto \text{right } (n, 1) \\ \text{right } (n + 1, m) & \mapsto \text{right } (n, n \times m) \\ \text{right } (0, m) & \mapsto \text{left } m \end{array} \right.$$

Recursion through a Trace

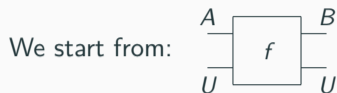


The recurring example: g :

$$\begin{cases} \mathbb{N} \uplus (\mathbb{N} \times \mathbb{N}) & \rightarrow & \mathbb{N} \uplus (\mathbb{N} \times \mathbb{N}) \\ \text{left } n & \mapsto & \text{right } (n, 1) \\ \text{right } (n + 1, m) & \mapsto & \text{right } (n, n \times m) \\ \text{right } (0, m) & \mapsto & \text{left } m \end{cases}$$

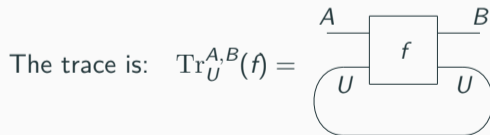


Recursion through a Trace



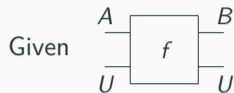
The recurring example: g :

$$\begin{cases} \mathbb{N} \uplus (\mathbb{N} \times \mathbb{N}) & \rightarrow \mathbb{N} \uplus (\mathbb{N} \times \mathbb{N}) \\ \text{left } n & \mapsto \text{right } (n, 1) \\ \text{right } (n+1, m) & \mapsto \text{right } (n, n \times m) \\ \text{right } (0, m) & \mapsto \text{left } m \end{cases}$$



And $\text{Tr}_{\mathbb{N} \uplus \mathbb{N}}^{\mathbb{N}, \mathbb{N}}(g)(n) = n!$.

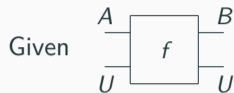
The Quantum Case



The canonical trace formula is:

$$\text{Tr}(f) = f_{AB} + \sum_n f_{UB} f_{UU}^n f_{AU}$$

The Quantum Case



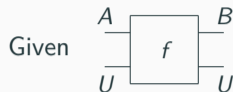
The canonical trace formula is:

$$\text{Tr}(f) = f_{AB} + \sum_n f_{UB} f_{UU}^n f_{AU}$$

This is a trace for unitaries between **finite dimension** Hilbert space.

However:

The Quantum Case



The canonical trace formula is:

$$\text{Tr}(f) = f_{AB} + \sum_n f_{UB} f_{UU}^n f_{AU}$$

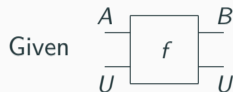
This is a trace for unitaries between **finite dimension** Hilbert space.

However:

Lemma (Towards no go)

The canonical trace on infinite dimension Hilbert spaces is not a trace.

The Quantum Case



The canonical trace formula is:

$$\mathrm{Tr}(f) = f_{AB} + \sum_n f_{UB} f_{UU}^n f_{AU}$$

This is a trace for unitaries between **finite dimension** Hilbert space.

However:

Lemma (Towards no go)

The canonical trace on infinite dimension Hilbert spaces is not a trace.

My take:

Conjecture: No go

There is **no suitable trace** for (sub)unitaries between infinite dimension Hilbert spaces.

Until we meet again

Is quantum recursion not possible?

If you ask people in the community:

Until we meet again

Is quantum recursion not possible?

If you ask people in the community:

- Classical recursion allows **non termination**.

Is quantum recursion not possible?

If you ask people in the community:

- Classical recursion allows **non termination**.
- *“Non-terminating quantum-controlled program do not make sense.”*

Is quantum recursion not possible?

If you ask people in the community:

- Classical recursion allows **non termination**.
- *“Non-terminating quantum-controlled program do not make sense.”*
- But no mathematical account of this point yet.

Until we meet again

Is quantum recursion not possible?

If you ask people in the community:

- Classical recursion allows **non termination**.
- *“Non-terminating quantum-controlled program do not make sense.”*
- But no mathematical account of this point yet.

Termination needs to be ensured!

Until we meet again

Is quantum recursion not possible?

If you ask people in the community:

- Classical recursion allows **non termination**.
- *“Non-terminating quantum-controlled program do not make sense.”*
- But no mathematical account of this point yet.

Termination needs to be ensured!

A technique called **guarded** recursion.

Guarded Quantum Recursion

Let H be the Hilbert space for lists of qubits.

Guarded Quantum Recursion

Let H be the Hilbert space for lists of qubits.

Decompose H as the following sequence:

$$H_0 \longleftarrow H_1 \longleftarrow H_2 \longleftarrow H_3 \longleftarrow \dots$$

where H_n is the lists of qubits up to size n .

Guarded Quantum Recursion

Let H be the Hilbert space for lists of qubits.

Decompose H as the following sequence:

$$H_0 \longleftarrow H_1 \longleftarrow H_2 \longleftarrow H_3 \longleftarrow \dots$$

where H_n is the lists of qubits up to size n .

Then, a function is described as a family of functions $\{f_i: H_i \rightarrow H_i\}$.

Guarded Quantum Recursion

Let H be the Hilbert space for lists of qubits.

Decompose H as the following sequence:

$$H_0 \longleftarrow H_1 \longleftarrow H_2 \longleftarrow H_3 \longleftarrow \dots$$

where H_n is the lists of qubits up to size n .

Then, a function is described as a family of functions $\{f_i: H_i \rightarrow H_i\}$.

→ sufficiently **nice**: natural transformation.

Guarded Quantum Recursion

Let H be the Hilbert space for lists of qubits.

Decompose H as the following sequence:

$$H_0 \longleftarrow H_1 \longleftarrow H_2 \longleftarrow H_3 \longleftarrow \dots$$

where H_n is the lists of qubits up to size n .

Then, a function is described as a family of functions $\{f_i: H_i \rightarrow H_i\}$.

→ sufficiently **nice**: natural transformation.

In this setting, admissible fixed points [Lemonnier2024].

Guarded Quantum Recursion

Let H be the Hilbert space for lists of qubits.

Decompose H as the following sequence:

$$H_0 \longleftarrow H_1 \longleftarrow H_2 \longleftarrow H_3 \longleftarrow \dots$$

where H_n is the lists of qubits up to size n .

Then, a function is described as a family of functions $\{f_i: H_i \rightarrow H_i\}$.

→ sufficiently **nice**: natural transformation.

In this setting, admissible fixed points [Lemonnier2024].

However, the expressivity is limited.

Conclusion

- ◆ Reversibility does not prevent expressivity.

Conclusion

- ◆ Reversibility does not prevent expressivity.
- ◆ Does not naturally generalise to the reversible quantum case.

Conclusion

- ◆ Reversibility does not prevent expressivity.
- ◆ Does not naturally generalise to the reversible quantum case.
- ◆ Do you think **quantum recursion** makes sense?

- ◆ Reversibility does not prevent expressivity.
- ◆ Does not naturally generalise to the reversible quantum case.
- ◆ Do you think **quantum recursion** makes sense?

Thank you!